UKStay Database Management Report

Table of Contents

Business Context and Database Design

Our hypothetical start-up company, UKStay, operates as a platform exclusively in the United Kingdom (UK), connecting hosts who rent out their properties to guests looking for accommodation. The database looks to maximise revenue through its core functions, implemented in the mini world: user, listing and booking management, payment and refund processing, review systems, and user activity tracking.

Based on this, we have developed a comprehensive database design. To visualise the database’s structure, we created an Entity-Relationship (ER) Diagram, illustrating key entities, their attributes and relationships. We also prioritised the use of Crow’s Foot notation over Chen’s, for a better representation of entity connections, cardinalities, and smoother scalability.

Figure 1 visually represents the structure of our database, while Table 1 complements it by providing a detailed breakdown of each entity and its relationships.

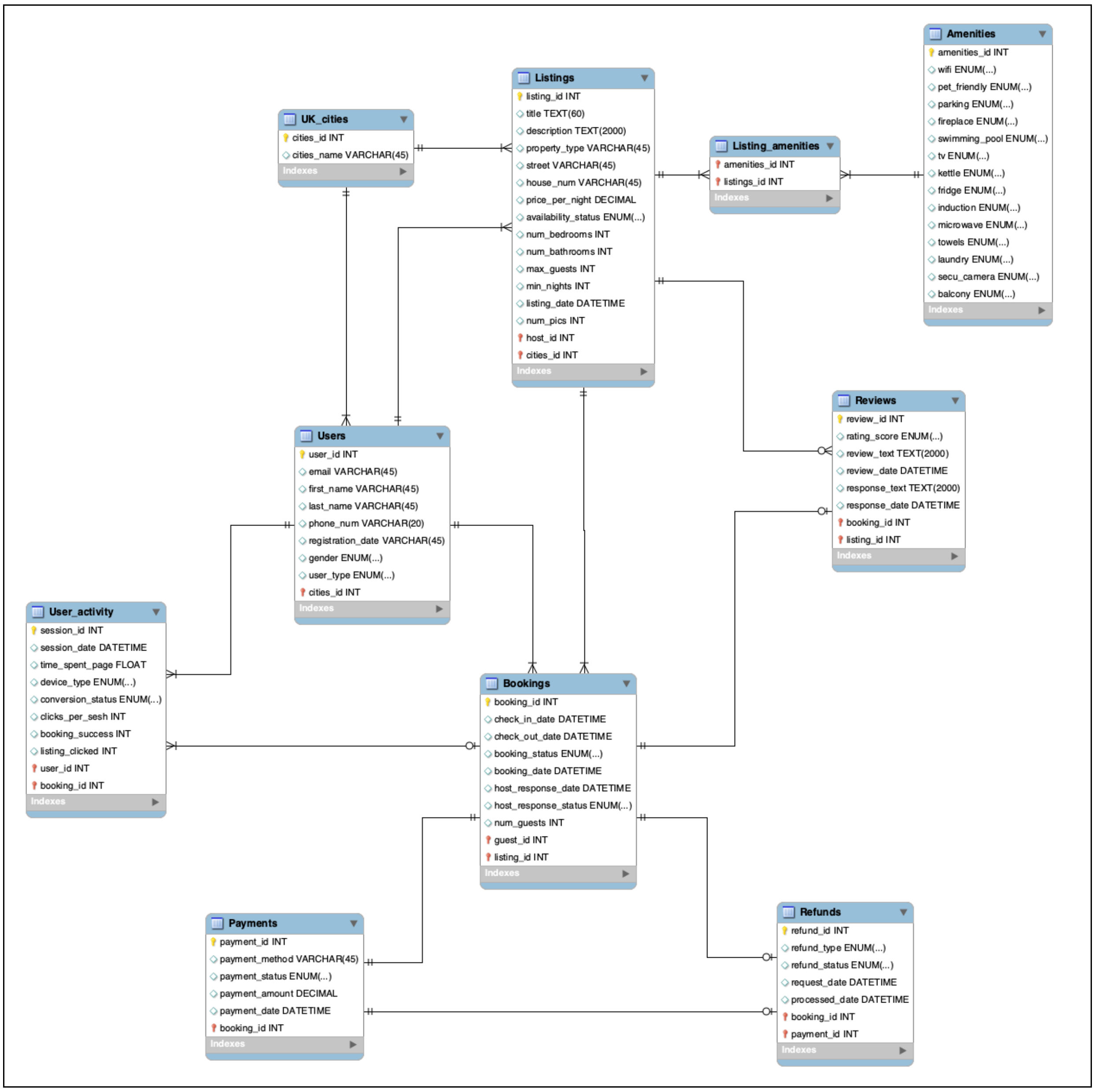


Figure 1: UKStay's ER Diagram

Entity	Attribute (Primary Key, **Foreign Key**)	Description	Relationships – optional relationship has the word 'can' mentioned
Users - store users' personal data, regardless of their type	user_id	Unique identifier for each user	
	cities_id	References UK_cities, indicating the user's city	1 city has one or multiple users (1:M)
	email	User's email address	

	first_name	User’s first name	
	last_name	User’s last name	
	phone_num	User’s contact number	
	registration_date	Date the user registered	
	gender	User’s gender	
	user_type	Defines the user as a host or guest	
Listings – contain detailed information about properties for rent	listing_id	Unique identifier for each listing	
	host_id	References Users, identifying the host	1 user/host owns one or multiple listings (1:M)
	cities_id	References UK_cities, indicating the listing’s city	1 city has one or multiple listings (1:M)
	title	Title of the listing	
	description	Description of the listing	
	property_type	Type of property (e.g., apartment, house)	
	street	Street address	
	house_num	House number	
	price_per_night	Cost per night	
	availability_status	Whether the listing is available	
	num_bedrooms	Number of bedrooms	
	num_bathrooms	Number of bathrooms	
	max_guests	Maximum guests allowed	
	min_nights	Minimum required nights	
	listing_date	Date the listing was created	
	num_pics	Number of pictures uploaded	
Bookings – manage reservation details	booking_id	Unique identifier for each booking	
	guest_id	References Users, identifying the guest	1 guest makes one or multiple bookings (1:M)

	listing_id	References Listings, identifying the property booked	1 listing has one or multiple bookings (1:M)
	check_in_date	Check-in date	
	check_out_date	Check-out date	
	booking_status	Status (e.g., confirmed, canceled)	
	booking_date	Date the booking was made	
	host_response_date	Date the host responded	
	host_response_status	Response status (e.g., accepted, declined)	
	num_guests	Number of guests	
User_activity – track user interactions each time they are on the website	<u>session_id</u>	Unique session identifier	
	user_id	References Users, identifying the user	1 user has one or multiple activity records (1:M)
	booking_id	References Bookings if a session resulted in a booking	1 booking CAN result from many user activities (1:M)
	booking_success	Whether the session led to a booking	
	listing_clicked	Number of clicked listings	
	session_date	Date of the session	
	time_spent_page	Time spent (in minutes)	
	device_type	Device used (mobile, desktop)	
	conversion_status	Status of conversion (e.g., returning user, new user)	
	clicks_per_sesh	Number of clicks per session	
Payments – track user interactions each time they are on the website	<u>payment_id</u>	Unique identifier for each payment	

	booking_id	References Bookings, linking the payment to a booking	1 booking has exactly 1 payment (1:1)
	payment_method	Method (e.g., credit card, PayPal)	
	payment_status	Status (e.g., completed, pending)	
	payment_amount	Amount paid	
	payment_date	Date of payment	
Refunds – manage refund requests associated with bookings	refund_id	Unique identifier for each refund	
	booking_id	References Bookings, linking refund to a booking	1 booking CAN allow 1 refund (1:1)
	payment_id	References Payments, linking refund to a payment	1 payment CAN allow 1 refund (1:1)
	refund_type	Type of refund (full, partial)	
	request_date	Date refund was requested	
	processed_date	Date refund was processed	
Reviews – collect feedback and ratings for listings	review_id	Unique identifier for each review	
	booking_id	References Bookings, linking review to a booking	1 booking allows 1 review (1:1)
	listing_id	References Listings, linking review to a listing	1 listing CAN have multiple reviews (1:M)
	rating_score	Numeric rating of the experience	
	review_text	Text feedback	
	review_date	Date review was posted	
	response_text	Host’s response to the review	
	response_date	Date of host’s response	

Amenities – store information about available amenities	<u>amenities_id</u>	Unique identifier for each amenity	
	wifi, pet_friendly, parking, fireplace, swimming_pool, tv, kettle, fridge, induction, microwave, towels, laundry, secu_camera, balcony	Individual amenities	
Listing_amenities – connect listings with their associated amenities. Purpose of this table is to join as this is (M:N) relationship	**amenities_id**	References Amenities	1 amenity is mentioned by various listings (1:M)
	listing_id	References Listings	1 listing will have multiple amenities (1:M)
UK_cities – store UK city names	<u>cities_id</u>	Unique identifier for each city	
	cities_name	Name of the city	

Table 1: Breakdown of UKStay's ER Diagram

Several assumptions were also made for simplification purposes:

- **Geographic location**: The platform operates exclusively within 30 main UK cities, meaning all users and properties are within those areas, to focus on a specific market.

- **User types**: We limit our users to being either one of the two user types – a guest (who makes bookings) or a host (who rents out properties) while maintaining the same user_id throughout. This allows for a unified account system, where interactions across both roles are simplified.

- **Amenities limitation**: Amenities were limited to 14 key attributes, mostly related to outdoor and kitchen services, although real-world platforms may offer hundreds of options. We have assumed that safety-related amenities (such as smoke alarms) should be mandatory for all listings.

- **Booking and payment integration**: We assume that a booking automatically triggers a payment process and that payments are processed in full upfront. While real-world platforms may involve multiple payment methods or installments, this assumption streamlines the transaction flow.

Data Implementation and Synthetic Data Generation

In this section, we will look at defining and implementing the relational database schema for UKStay, based on our ER Diagram, and further on generate synthetic data to populate the database.

To implement our schema, we used SQLite, specifically SQL Data Definition Language (DDL), to define tables and their relationships. This approach is essential as it enforces constraints and maintains referential integrity across the database. This, in turn, supported our initiative to make our project scalable and avoid data redundancy. Table 2 presents the data types across entities, with explanations as to their selection.

Data Type	Attribute (Table)	Explanation
INTEGER PRIMARY KEY	user_id (Users), listing_id (Listings), booking_id (Bookings), payment_id (Payments), session_id (User_activity), refund_id (Refunds), review_id (Reviews), amenities_id (Amenities)	Allow uniqueness and non-null values. Incrementation is manually added to avoid duplicates within the same table or the presence of only the first row of data.
INTEGER PRIMARY KEY AUTOINCREMENT	cities_id (UK_cities)	Ensure each city has a unique identifier, automatically incrementing for new entries.
TEXT	email, first_name, last_name (Users), title, description, property_type, street, house_num (Listings), payment_method (Payments),	Accommodates variable lengths and alphanumeric values.

	review_text, response_text (Reviews), cities_name (UK_cities)	
TEXT	phone_num (Users)	Preserve UK number formatting.
TEXT CHECK	gender, user_type (Users), availability_status (Listings), booking_status, host_response_status (Bookings), device_type, conversion_status (User_activity), payment_status (Payments), refund_type, refund_status (Refunds), wifi, pet_friendly, parking, fireplace, swimming_pool, tv, kettle, fridge, induction, microwave, towels, laundry, secu_camera, balcony (Amenities)	Enforce allowed values for categorical attributes (such as 'Male' or 'Female' for gender).
TEXT	registration_date (Users), listing_date (Listings), check_in_date, check_out_date, booking_date, host_response_date (Bookings), session_date (User_activity), payment_date (Payments), request_date, processed_date (Refunds), review_date, response_date (Reviews)	Ease readability and sorting, as SQLite lacks a DATE type.
INTEGER	num_bedrooms, num_bathrooms, max_guests, min_nights, num_pics (Listings), num_guests (Bookings), clicks_per_sesh, booking_success, listing_clicked (User_activity), rating_score (Reviews)	Store whole numbers.
DECIMAL	price_per_night (Listings), payment_amount (Payments), refund_amount (Refunds), time_spent_page (User_activity)	Precise storage of monetary and time values.
FOREIGN KEY	cities_id (Users), host_id (Listings), guest_id (Bookings), listing_id (Bookings, Reviews), user_id (User_activity), booking_id (Payments), payment_id (Refunds), amenities_id (Listing_Amenities)	Ensure referential integrity with related tables.

Table 2: Data type for each entity in UKStay's schema

While developing our schema, we considered separate tables for host and guest users but opted against it to maintain consistency. Our current design, normalized to third normal form, minimizes redundancy and allows for easy expansion without major structural changes. Future enhancements could include supporting international users and adding tables for currency and phone extensions.

Following this, we populated the database with synthetic data using Python's *Faker* library, allowing us to generate structured, predefined data that aligned with our database constraints. With the SQLite tables, we ensured data consistency through foreign key constraints, which allowed the maintenance of relationships even as the data was being generated, as well as using *for* and *if* loops within the code, helping conserve the specific sequence of events across datasets.

Variety was introduced by combining *Faker* with Python's *random* library, as each covered different aspects of randomisation: the former generates text-based data whilst the latter focuses on categorical selection, numerical values and probability distributions. The extended code is provided in Appendix A, which details the data generation process. Altogether, this formed a well-balanced dataset, in which about 500 rows of data were generated per table, excepting the UK_cities table, limited to only 30, to allow for city-specific insights.

A main issue that appeared was maintaining logical sequences when random dates were being generated, which was solved with the use of *datetime* and *timedelta* libraries. Another challenge involved handling refund dependencies, which were tied to data from other tables and had to be restricted to specific bookings. This required the development of conditional statements and predefined rules, along with randomisation, to assign refunds appropriately.

Validation and accuracy checks were also implemented to ensure the respect of both integrity and correctness standards, presented in Appendix B. Specifically, these included data integrity, logical data validation (i.e. check-in date cannot be before check-out date), cross-table consistency and business rules' compliance (guests cannot put out properties, for instance).

Business Insights Reports

We were then able to generate business insights reports, to extract meaningful patterns from the dataset related to revenue maximisation. These insights used SQL queries and visualisations made in Tableau, divided into revenue, city popularity and market analyses.

In our revenue analysis, we focused on evaluating the performance of different property types. By joining the booking, listing, and payment tables, we grouped and aggregated data by property type to analyse total bookings and revenue. To ensure accuracy, we counted only bookings with a "paid" payment status, preventing inflated numbers while still considering the total transaction value across all payment statuses to reflect overall market demand. We then calculated each property type's share of paid bookings, providing insights into market dominance (Figure 2).

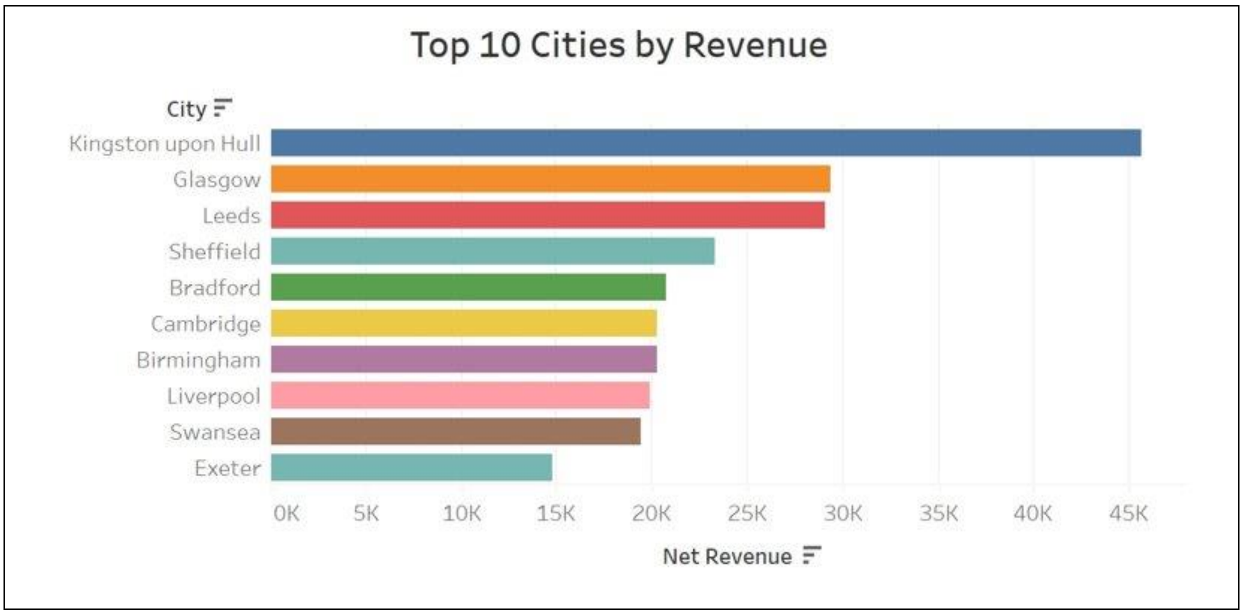
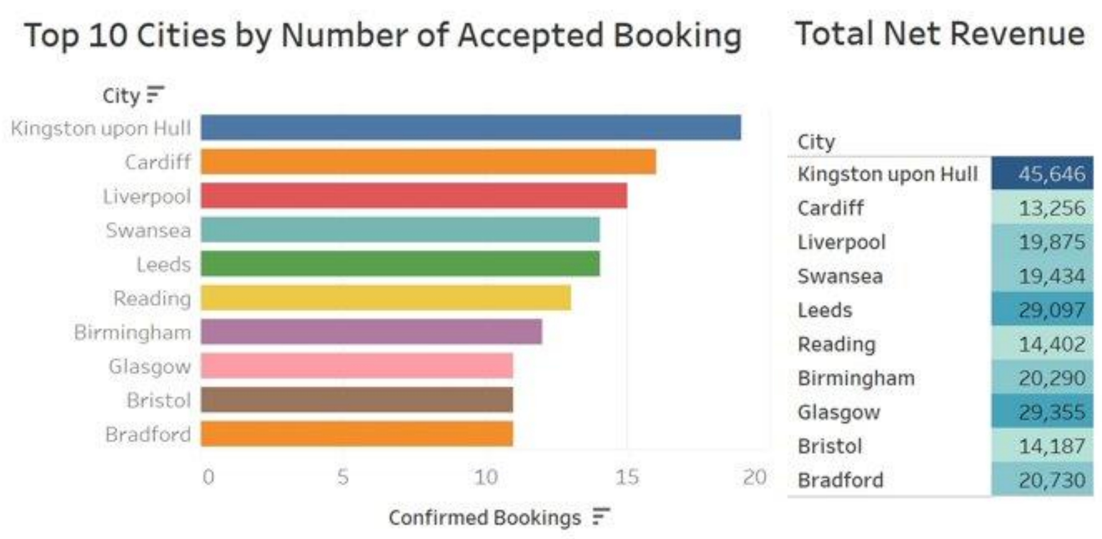


Figure 2: Cities with the highest net revenue

For city popularity, we analysed booking distribution across cities by linking the UK cities, listing, booking, and payment tables to extract total bookings and total revenue for each location (Figure 3). We sorted cities by revenue and booking volume, ensuring each record corresponds to actual transactions. This analysis helped us identify high-performing and emerging markets.

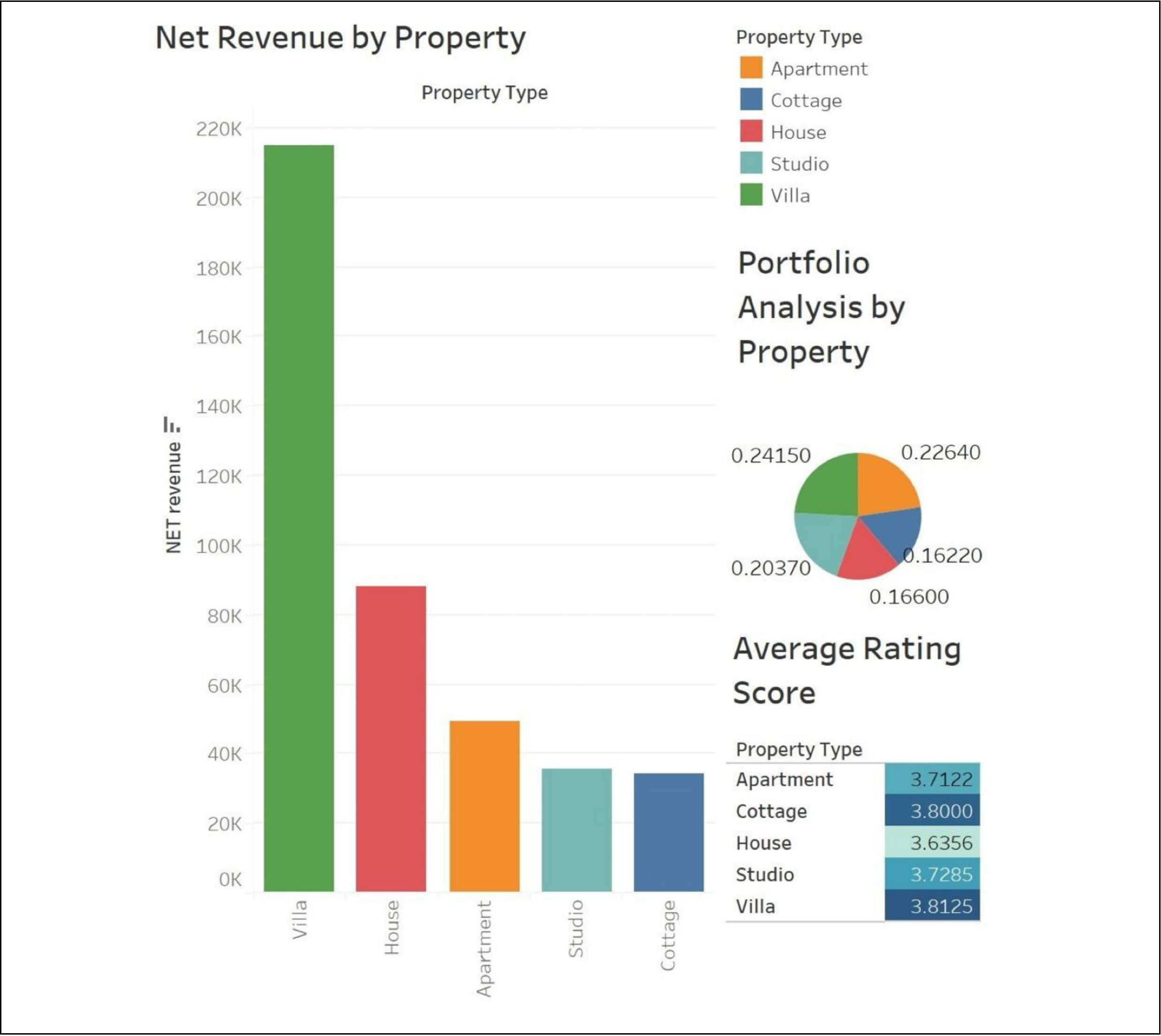


Total Net Revenue

City	
Kingston upon Hull	45,646
Cardiff	13,256
Liverpool	19,875
Swansea	19,434
Leeds	29,097
Reading	14,402
Birmingham	20,290
Glasgow	29,355
Bristol	14,187
Bradford	20,730

Figure 3: Comparison between cities with most confirmed bookings and net revenue

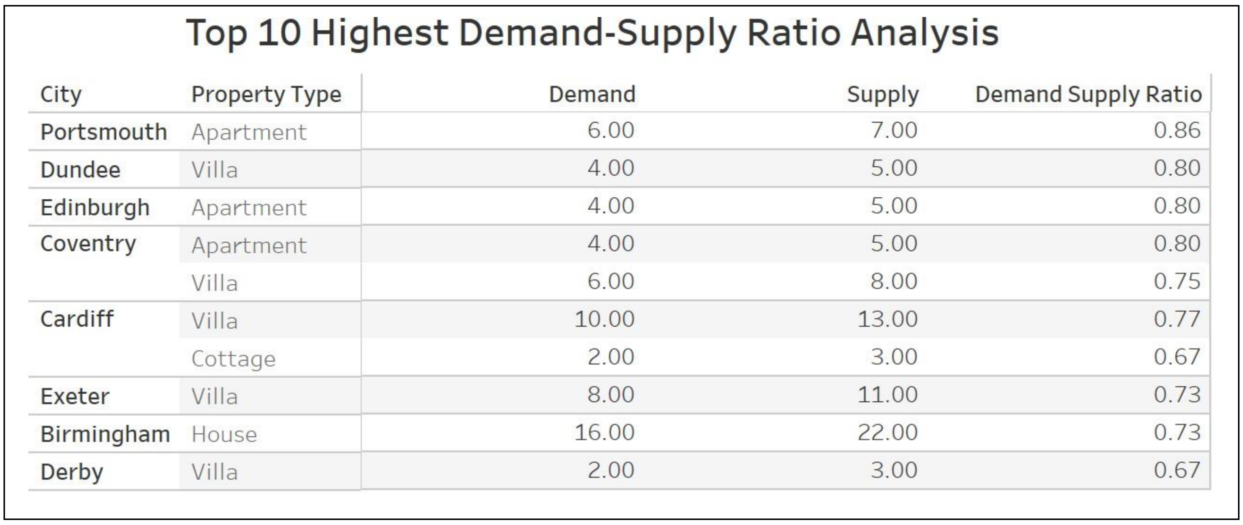
In our market analysis, we evaluated supply and demand dynamics and prime property distribution to identify key trends across cities and property types. Using data from listing, booking, payment, and city tables, we analysed booking demand, property availability and revenue performance (Figure 4).



Property Type	
Apartment	3.7122
Cottage	3.8000
House	3.6356
Studio	3.7285
Villa	3.8125

Figure 4: Net revenue, portfolio analysis and average rating score by property

Additionally, to assess the demand-supply balance, we calculated the ratio of confirmed bookings to total listings, identifying high-demand markets and oversaturated areas. By evaluating net revenue per property, we uncovered high-yielding property types.


Top 10 Highest Demand-Supply Ratio Analysis

City	Property Type	Demand	Supply	Demand Supply Ratio
Portsmouth	Apartment	6.00	7.00	0.86
Dundee	Villa	4.00	5.00	0.80
Edinburgh	Apartment	4.00	5.00	0.80
Coventry	Apartment	4.00	5.00	0.80
	Villa	6.00	8.00	0.75
Cardiff	Villa	10.00	13.00	0.77
	Cottage	2.00	3.00	0.67
Exeter	Villa	8.00	11.00	0.73
Birmingham	House	16.00	22.00	0.73
Derby	Villa	2.00	3.00	0.67

Table 3: Demand-supply ratio per city and property type

Conclusion

To enhance customer retention and maximise revenue, UKStay will enhance its refund policy where partial refunds are based on cancellation timing, offering cashbacks and discounts to improve guest satisfaction. Additionally, real-time user tracking will allow the marketing team to analyse trends and personalise recommendations.

To support scalability, UKStay will transition to a cloud-based database, ensuring faster data processing and seamless platform operations. A sustainable data analysis system could enable dynamic pricing strategies, balancing revenue and occupancy rates based on real-time supply-demand insights. By leveraging data-driven decision-making, UKStay can refine marketing efforts, optimise property pricing, and improve host and guest experiences.

Appendix

Appendix A: Extended code for UKStay's database implementation and synthetic data generation

Part A: UK_cities table

```
import sqlite3
from faker import Faker
import random

# Connect to SQLite database
conn = sqlite3.connect("mydm.db")
cursor = conn.cursor()

# Create UK_cities table
cursor.execute('''
CREATE TABLE IF NOT EXISTS UK_cities (
cities_id INTEGER PRIMARY KEY AUTOINCREMENT,
cities_name TEXT
);
''')

# Sample list of UK cities
cities_name = [
"London", "Birmingham", "Manchester", "Liverpool", "Leeds",
"Sheffield", "Bristol", "Newcastle upon Tyne", "Nottingham", "Leicester",
"Coventry", "Bradford", "Southampton", "Derby", "Stoke-on-Trent",
"Wolverhampton", "Plymouth", "Reading", "Kingston upon Hull", "Belfast",
"Edinburgh", "Glasgow", "Cardiff", "Swansea", "Aberdeen",
"Dundee", "Portsmouth", "York", "Exeter", "Cambridge"
]

# Initialise cities_id counter
cities_id_counter = 1

for city in cities_name:
try:
cursor.execute("INSERT INTO UK_cities (cities_id, cities_name) VALUES (?, ?)",
(cities_id_counter, city))
cities_id_counter += 1
except sqlite3.IntegrityError:
print(f"City '{city}' already exists in the database.")
```

```
conn.commit()
conn.close()
print("UK_cities table populated with fake data.")
```

Part B: Users table

```
import sqlite3
import random
from faker import Faker

# Initialise Faker
fake = Faker()

# Connect to the database
conn = sqlite3.connect("mydm.db")
cursor = conn.cursor()

# Create Users table if it doesn't exist
cursor.execute('''
CREATE TABLE IF NOT EXISTS Users (
user_id INTEGER PRIMARY KEY,
email TEXT,
first_name TEXT,
last_name TEXT,
phone_num TEXT,
registration_date TEXT,
gender TEXT CHECK (gender IN ('Male', 'Female')),
user_type TEXT CHECK (user_type IN ('Guest', 'Host')),
cities_id INTEGER NOT NULL,
FOREIGN KEY (cities_id) REFERENCES UK_cities (cities_id)
);
''')
#gender - 0: Male, 1: Female
#user_type - 0: Guest, 1: Host

# Function to generate a random phone number
def generate_phone_number():
```

```
first_digit = 0
remaining_digits = [random.randint(0, 9) for _ in range(9)]
return str(first_digit) + ''.join(map(str, remaining_digits))

from datetime import date, timedelta
# Get today's date
today = date.today()
# Calculate the date one year ago
one_year_ago = today - timedelta(days=365)
# Generate a date within the last year
date_within_last_year = fake.date_between(start_date=one_year_ago, end_date=today)

# Generate users
users_data = []
# Initialise user_id counter
user_id_counter = 1
cursor.execute("SELECT cities_id FROM UK_cities")
available_cities_ids = [row[0] for row in cursor.fetchall()]

for _ in range(500):
gender_choice = random.choice(["M", "F"])
first_name = fake.first_name_male() if gender_choice =="M" else fake.first_name_female()
last_name = fake.last_name()
email = f"{first_name[0].lower()}.{last_name.lower()}@{fake.domain_name()}"
phone_num = generate_phone_number()
registration_date = fake.date_between(start_date=one_year_ago, end_date=today).isoformat()
gender = 'Male' if gender_choice == "M" else 'Female'
user_type = 'Guest' if random.choice([0, 1]) == 0 else 'Host'
cities_id = random.choice(available_cities_ids)
users_data.append((user_id_counter, email, first_name, last_name, phone_num,
registration_date, gender, user_type, cities_id))
user_id_counter += 1

# Insert user data with manually managed user_id and error handling
try:
cursor.executemany("""
INSERT INTO Users (user_id, email, first_name, last_name, phone_num, registration_date,
gender, user_type, cities_id)
VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)
""", users_data)
conn.commit()
#except sqlite3.IntegrityError:
# print(f"Error inserting user: {user_data}. user_id might already exist.")
#except Exception as e:
# print(f"An unexpected error occurred: {e}")
except sqlite3.IntegrityError as e:
if "UNIQUE constraint failed" in str(e): # Check for the specific error message
print(f"Error inserting user: {users_data}. user_id might already exist.")
else:
raise # Re-raise other exceptions
# Close the connection
conn.close()

print("Users table populated with fake data.")
```

Part C: Listings table

```
import sqlite3
import random
from faker import Faker

# Initialise Faker
fake = Faker()

# Connect to the database
conn = sqlite3.connect("mydm.db")
cursor = conn.cursor()

# Create Listings table if it doesn't exist
cursor.execute('''
CREATE TABLE IF NOT EXISTS Listings (
listing_id INTEGER PRIMARY KEY,
title TEXT,
description TEXT,
property_type TEXT,
street TEXT,
house_num TEXT,
price_per_night DECIMAL,
availability_status TEXT CHECK (availability_status IN ('available', 'booked')),
num_bedrooms INTEGER,
num_bathrooms INTEGER,
max_guests INTEGER,
min_nights INTEGER,
listing_date TEXT,
num_pics INTEGER,
host_id INTEGER NOT NULL,
cities_id INTEGER NOT NULL,
FOREIGN KEY (host_id) REFERENCES Users (user_id),
FOREIGN KEY (cities_id) REFERENCES UK_cities (cities_id)
);
''')

# Initialise listing_id counter
listing_id_counter = 1

# Generate listings data
listings_data = []
cursor.execute("SELECT user_id FROM Users WHERE user_type = 'Host'") # Get host IDs
available_host_ids = [row[0] for row in cursor.fetchall()]
cursor.execute("SELECT cities_id FROM UK_cities")
available_cities_ids = [row[0] for row in cursor.fetchall()]

property_types = ["Apartment", "House", "Studio", "Cottage", "Villa"]

property_keywords = {
"Apartment": ["cozy", "modern", "city view", "studio", "balcony", "spacious", "well-equipped",
"central location", "quiet neighborhood"],
"House": ["spacious", "family-friendly", "garden", "fireplace", "garage", "private", "comfortable",
"large kitchen", "multiple bedrooms"],
"Studio": ["compact", "efficient", "minimalist", "city center", "affordable", "stylish", "open-plan",
"great for students", "close to amenities"],
"Cottage": ["charming", "rustic", "cozy", "garden", "countryside", "traditional", "fireplace",
"peaceful", "scenic views"],
"Villa": ["luxury", "spacious", "private pool", "garden", "sea view", "modern", "fully equipped",
"exclusive", "breathtaking views"]
}

property_bedrooms = {
"Apartment": (1, 3),
"House": (2, 5),
"Studio": (1),
"Cottage": (1, 3),
"Villa": (3, 6)
}

from datetime import date, timedelta
# Get today's date
today = date.today()
# Calculate the date one year ago
one_year_ago = today - timedelta(days=365)
# Generate a date within the last year
date_within_last_year = fake.date_between(start_date=one_year_ago, end_date=today)

def get_max_guests(num_bedrooms):
if num_bedrooms in range(1, 6):
return {1: 2, 2: 4, 3: 6, 4: 8, 5: 10}.get(num_bedrooms)
else:
return 15 # Default max_guests if num_bedrooms is outside the range

def get_num_bathrooms(num_bedrooms):
if num_bedrooms in range(1, 6):
return {1: 1, 2: 1, 3: 2, 4: 3, 5: 4}.get(num_bedrooms)
else:
return 5

for _ in range(500):
property_type = random.choice(property_types)
keywords = property_keywords.get(property_type, []) # Get keywords for property type
```

```
title = f"{random.choice(keywords)} {property_type}"
description = f"This {property_type} is {random.choice(keywords)} and features a
{random.choice(keywords)}. It is located in a {random.choice(keywords)} area and is perfect for
{random.choice(['families', 'couples', 'solo travelers'])}."
street = fake.street_name()
house_num = str(random.randint(1, 100))
availability_status = random.choice(['available', 'booked'])

bedroom_range = property_bedrooms.get(property_type)
if isinstance(bedroom_range, tuple):
if len(bedroom_range) == 2: # For ranges (min, max)
num_bedrooms = random.randint(bedroom_range[0], bedroom_range[1])
elif len(bedroom_range) == 1: # For fixed values (single tuple element)
num_bedrooms = bedroom_range[0]
else: # If property_type not in property_bedrooms, use a default
num_bedrooms = 2
max_guests = get_max_guests(num_bedrooms)
num_bathrooms = get_num_bathrooms(num_bedrooms)
min_nights = random.randint(1, 3)

base_price = 50 # Starting price
price_per_night = base_price + \
int(min_nights) * 5 + \
int(max_guests) * 10 + \
int(num_bedrooms) * 20
price_per_night = round(price_per_night, 2)

listing_date = fake.date_between(start_date=one_year_ago, end_date=today)
num_pics = random.randint(1, 10)
host_id = random.choice(available_host_ids)
cities_id = random.choice(available_cities_ids)

listings_data.append((listing_id_counter,title, description, property_type, street, house_num,
price_per_night,
availability_status, num_bedrooms, num_bathrooms,
max_guests, min_nights, listing_date, num_pics, host_id, cities_id))
listing_id_counter += 1
```

```
# Insert listings data
try:
cursor.executemany("""
INSERT INTO Listings (listing_id, title, description, property_type, street, house_num,
price_per_night,
availability_status, num_bedrooms, num_bathrooms,
max_guests, min_nights, listing_date, num_pics, host_id, cities_id)
VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
""", listings_data)
listing_id_counter += 1
conn.commit()
except sqlite3.IntegrityError:
print(f"Error inserting user: {listings_data}. listing_id might already exist.")
except Exception as e:
print(f"An unexpected error occurred: {e}")

# Close the connection
conn.close()

print("Listings table populated with fake data.")
```

Part D: Bookings table

```
import sqlite3
import random
from faker import Faker
from datetime import datetime, timedelta

# Initialise Faker
fake = Faker()

# Connect to the database
conn = sqlite3.connect("mydm.db")
cursor = conn.cursor()

# Create Bookings table
cursor.execute('''
CREATE TABLE IF NOT EXISTS Bookings (
booking_id INTEGER PRIMARY KEY,
check_in_date TEXT,
check_out_date TEXT,
booking_status TEXT CHECK (booking_status IN ('confirmed', 'cancelled by host', 'cancelled by
guest', 'pending')),
booking_date TEXT,
host_response_date TEXT,
host_response_status TEXT CHECK (host_response_status IN ('accepted', 'rejected', 'pending')),
num_guests INTEGER,
guest_id INTEGER NOT NULL,
listing_id INTEGER NOT NULL,
FOREIGN KEY (guest_id) REFERENCES Users (user_id),
FOREIGN KEY (listing_id) REFERENCES Listings (listing_id)
);
''')

# Get available guest and listing IDs
cursor.execute("SELECT user_id FROM Users WHERE user_type = 'Guest'")
available_guest_ids = [row[0] for row in cursor.fetchall()]
cursor.execute("SELECT listing_id FROM Listings")
available_listing_ids = [row[0] for row in cursor.fetchall()]

# Generate bookings data
bookings_data = []
booking_id_counter = 1
booked_listings = set() # Keep track of booked listings to avoid double bookings

today = date.today()
one_year_ago = today - timedelta(days=365)

for _ in range(500):
listing_id = random.choice(available_listing_ids)

# Fetch listing_date for the selected listing_id
cursor.execute("SELECT listing_date FROM Listings WHERE listing_id = ?", (listing_id,))
listing_date_str = cursor.fetchone()[0]
listing_date = datetime.fromisoformat(listing_date_str).date()
```

```
# Generate random dates within a reasonable range
check_in_date = fake.date_between(start_date=one_year_ago, end_date=today)
booking_date = fake.date_between(start_date=one_year_ago, end_date=min(check_in_date,
today))
check_out_date = check_in_date + timedelta(days=random.randint(1, 21))

cursor.execute("SELECT availability_status FROM Listings WHERE listing_id = ?", (listing_id,))
availability_status = cursor.fetchone()[0]

if availability_status == 'booked':
booking_status = random.choice(['confirmed'])
elif availability_status == 'available':
booking_status = random.choice(['cancelled by host', 'cancelled by guest','pending'])

# Map booking_status to host_response_status and host_response_date
response_logic = {
'confirmed': ('accepted', booking_date + timedelta(days=random.randint(1, 7))),
'pending': ('pending', None),
'cancelled by host': (random.choice(['rejected', 'pending']), booking_date +
timedelta(days=random.randint(1, 7))), # Always calculate host_response_date for 'cancelled by
host'
'cancelled by guest': (random.choice(['rejected', 'pending']), None),
}

# Get host_response_status and host_response_date
host_response_status, host_response_date = response_logic.get(booking_status, ('pending',
None))

if booking_status == 'cancelled by host' and host_response_status == 'pending':
host_response_date = None # Set to None if cancelled by host and pending

# Handle 'cancelled by guest' (add timedelta if response not pending)
if booking_status == 'cancelled by guest' and host_response_status != 'pending':
host_response_date = booking_date + timedelta(days=random.randint(1, 7))

# Prevent same guest from double-booking:
```

```
while (guest_id, check_in_date, check_out_date) in booked_listings:
guest_id = random.choice(available_guest_ids) # Choose a different guest
booked_listings.add((guest_id, check_in_date, check_out_date))

# 2. Prevent different guests from overlapping bookings:
while any(
(listing_id, existing_check_in, existing_check_out) in booked_listings
for existing_check_in in [check_in_date + timedelta(days=i) for i in range((check_out_date -
check_in_date).days +1)] # Enumerate booking range
for existing_check_out in [check_in_date + timedelta(days=i) for i in range((check_out_date -
check_in_date).days +1)] # Enumerate booking range
if existing_check_in <= existing_check_out # Ensure valid date range
):
check_in_date = fake.date_between(start_date=one_year_ago, end_date=today)
check_out_date = check_in_date + timedelta(days=random.randint(1, 21))

# Record the booking to prevent future conflicts:
for booking_day in range((check_out_date - check_in_date).days + 1):
booked_listings.add((listing_id, check_in_date + timedelta(days=booking_day), check_out_date))



# Generate num_guests considering only max_guests
cursor.execute("SELECT max_guests FROM Listings WHERE listing_id = ?", (listing_id,))
max_guests = cursor.fetchone()[0]
num_guests = random.randint(1, max_guests)

bookings_data.append((booking_id_counter, check_in_date.isoformat(),
check_out_date.isoformat(), booking_status,
booking_date.isoformat(), host_response_date.isoformat() if host_response_date else None,
host_response_status, num_guests, guest_id, listing_id))
booking_id_counter += 1

# Insert bookings data
try:
cursor.executemany("""
INSERT INTO Bookings (booking_id, check_in_date, check_out_date, booking_status,
booking_date,
host_response_date, host_response_status, num_guests, guest_id, listing_id)
VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
""", bookings_data)
conn.commit()
except sqlite3.IntegrityError as e:
if "UNIQUE constraint failed" in str(e): # Check for duplicate booking IDs
print(f"Error inserting data: booking_id might already exist.")
else:
raise # Re-raise other exceptions
except Exception as e:
print(f"An unexpected error occurred: {e}")

# Close the connection
conn.close()
print("Bookings table populated with fake data.")
```

Part E: Payments table

```
import sqlite3
import random
from faker import Faker
from datetime import datetime, timedelta

# Initialise Faker
fake = Faker()

# Connect to the database
conn = sqlite3.connect("mydm.db")
cursor = conn.cursor()

# Create Payments table
cursor.execute('''
CREATE TABLE IF NOT EXISTS Payments (
payment_id INTEGER PRIMARY KEY,
payment_method TEXT,
payment_status TEXT CHECK (payment_status IN ('paid', 'pending', 'failed')),
payment_amount DECIMAL,
payment_date TEXT,
booking_id INTEGER NOT NULL,
FOREIGN KEY (booking_id) REFERENCES Bookings (booking_id)
);
''')
#payment_status - 0: paid, 1: pending, 2: failed

# Get available booking IDs
cursor.execute("SELECT booking_id FROM Bookings")
available_booking_ids = [row[0] for row in cursor.fetchall()]

# Generate payments data
payments_data = []
payment_id_counter = 1
for booking_id in available_booking_ids:
payment_method = random.choice(["Credit/Debit Card", "PayPal", "Bank Transfer"])
payment_status = random.choice(['paid', 'pending', 'failed'])

# Fetch price_per_night and num_guests from Listings and Bookings tables
cursor.execute("""
SELECT L.price_per_night, B.num_guests
FROM Listings L
JOIN Bookings B ON L.listing_id = B.listing_id
WHERE B.booking_id = ?
""", (booking_id,))
price_per_night, num_guests = cursor.fetchone()

# Calculate payment amount
payment_amount = price_per_night * num_guests

# Generate payment date (around booking date)
cursor.execute("SELECT booking_date FROM Bookings WHERE booking_id = ?", (booking_id,))
booking_date_str = cursor.fetchone()[0]
booking_date = datetime.fromisoformat(booking_date_str).date()
payment_date = fake.date_between(start_date=booking_date - timedelta(days=3),
end_date=booking_date + timedelta(days=3))
```

```
# Set payment_status based on booking_status
cursor.execute("SELECT booking_status FROM Bookings WHERE booking_id = ?",
(booking_id,))
booking_status = cursor.fetchone()[0]
if booking_status == 'confirmed':
payment_status = 'paid'
elif booking_status == 'pending':
payment_status = 'pending'
else:
payment_status = random.choice(['failed', 'pending'])

payment_id_counter += 1
payments_data.append((payment_id_counter, payment_method, payment_status,
payment_amount, payment_date.isoformat(), booking_id))

# Insert payments data using executemany
try:
cursor.executemany("""
INSERT INTO Payments (payment_id, payment_method, payment_status, payment_amount,
payment_date, booking_id)
VALUES (?,?, ?, ?, ?, ?)
""", payments_data)
conn.commit()
except sqlite3.IntegrityError:
print(f"Error inserting user: {payments_data}. payment_id might already exist.")
except Exception as e:
print(f"An unexpected error occurred: {e}")
conn.close()

print("Payments table populated with fake data.")
```

Part F: Reviews table

```
import sqlite3
import random
from faker import Faker
from datetime import datetime, timedelta
```

```
# Initialise Faker
fake = Faker()

# Connect to the database
conn = sqlite3.connect("mydm.db")
cursor = conn.cursor()

# Create Reviews table (if not exists)
cursor.execute('''
CREATE TABLE IF NOT EXISTS Reviews (
review_id INTEGER PRIMARY KEY,
rating_score INTEGER,
review_text TEXT,
review_date TEXT,
response_text TEXT,
response_date TEXT,
booking_id INTEGER NOT NULL,
listing_id INTEGER NOT NULL,
FOREIGN KEY (booking_id) REFERENCES Bookings (booking_id),
FOREIGN KEY (listing_id) REFERENCES Listings (listing_id)
);
''')

# Get available booking and listing IDs
cursor.execute("SELECT DISTINCT listing_id FROM Bookings") # Get both booking_id and
listing_id
available_bookings_listings = [(row[0]) for row in cursor.fetchall()]

# Generate reviews data
reviews_data = []
review_id_counter = 1

for listing_id in available_bookings_listings:
num_reviews = random.randint(1,3)
for _ in range(num_reviews):
cursor.execute("SELECT booking_id FROM Bookings WHERE listing_id = ?", (listing_id,))
booking_ids_for_listing = [row[0] for row in cursor.fetchall()]
```

```
booking_id = random.choice(booking_ids_for_listing)
#rating_score = random.randint(1, 5)
# Define weights for each rating score
weights = [1, 2, 3, 4, 5] # Increase the weight for higher ratings
rating_score = random.choices(range(1, 6), weights=weights)[0]
if rating_score <= 2: # Consider ratings 1 or 2 as negative
sentiment = "negative"
else:
sentiment = "positive"
review_text = fake.paragraph(nb_sentences=3)

# Get check_out_date for the booking
cursor.execute("SELECT check_out_date FROM Bookings WHERE booking_id = ?",
(booking_id,))
check_out_date_str = cursor.fetchone()[0]
check_out_date = datetime.fromisoformat(check_out_date_str).date()

# Generate review_date after check_out_date
review_date = fake.date_between(start_date=check_out_date + timedelta(days=1),
end_date=check_out_date + timedelta(days=31))

# Get listing details for review generation
cursor.execute("SELECT property_type FROM Listings WHERE listing_id = ?", (listing_id,))
property_type = cursor.fetchone()[0]

# Generate review text based on sentiment
if sentiment == "positive":
review_text = f"I had a wonderful stay! The {property_type} was {random.choice(['clean',
'comfortable', 'well-equipped'])} and the location was {random.choice(['convenient', 'peaceful',
'close to amenities'])}"
else:
review_text = f"Unfortunately, my stay was not as expected. The {property_type} was
{random.choice(['dirty', 'uncomfortable', 'poorly equipped'])} and I encountered some issues with
the {random.choice(['cleanliness', 'facilities', 'communication'])}"

# Generate response text based on sentiment + host replies to guest
cursor.execute("SELECT guest_id FROM Bookings WHERE booking_id = ?", (booking_id,))
```

```
guest_id = cursor.fetchone()[0]
cursor.execute("SELECT host_id FROM Listings WHERE listing_id = ?", (listing_id,))
host_id = cursor.fetchone()[0]

response_chance = random.random()
if response_chance < 0.7 and guest_id != host_id:
if sentiment == "positive":
response_text = "Thank you for your kind words! We're so glad you enjoyed your stay."
elif sentiment == "negative":
response_text = "We apologize for the inconvenience you experienced. We'll take your feedback
into consideration to improve our services."
else:
response_text = None

response_date = fake.date_between(start_date=review_date + timedelta(days=1),
end_date=review_date + timedelta(days=14))
else:
response_text = None
response_date = None

reviews_data.append((review_id_counter, rating_score, review_text, review_date.isoformat(),
response_text, response_date.isoformat() if response_date else None, booking_id, listing_id))
review_id_counter += 1

# Insert reviews data
try:
cursor.executemany("""
INSERT INTO Reviews (review_id,rating_score, review_text, review_date, response_text,
response_date, booking_id, listing_id)
VALUES (?, ?, ?, ?, ?, ?, ?, ?)
""", reviews_data)
conn.commit()
except sqlite3.IntegrityError:
print(f"Error inserting user: {reviews_data}. review_id might already exist.")
except Exception as e:
print(f"An unexpected error occurred: {e}")
conn.close()
```

```
print("Reviews table populated with fake data.")
```

Part G: User_activity table

```
import sqlite3
import random
from faker import Faker
from datetime import datetime, timedelta

# Initialise Faker
fake = Faker()

# Connect to the database
conn = sqlite3.connect("mydm.db")
cursor = conn.cursor()

# Create User_activity table (if not exists)
cursor.execute('''
CREATE TABLE IF NOT EXISTS User_activity (
session_id INTEGER PRIMARY KEY,
session_date TEXT,
time_spent_page DECIMAL,
device_type TEXT CHECK (device_type IN ('mobile_phone', 'tablet', 'desktop')),
conversion_status TEXT CHECK (conversion_status IN ('repeating user', 'new user')),
clicks_per_sesh INTEGER,
user_id INTEGER NOT NULL,
booking_success INTEGER,
listing_clicked INTEGER,
booking_id INTEGER,
FOREIGN KEY (user_id) REFERENCES Users (user_id),
FOREIGN KEY (booking_id) REFERENCES Bookings (booking_id));
''')
#time in minutes

# Get available user, listing, and booking IDs
cursor.execute("SELECT user_id FROM Users")
available_user_ids = [row[0] for row in cursor.fetchall()]
```

```
# Get confirmed booking IDs
cursor.execute("SELECT booking_id FROM Bookings WHERE booking_status = 'confirmed'")
confirmed_booking_ids = [row[0] for row in cursor.fetchall()]

# Generate user activity data
user_activity_data = []
session_id_counter = 1
for _ in range(500):
user_id = random.choice(available_user_ids)
session_date = fake.date_this_year().isoformat()
time_spent_page = round(random.uniform(0.5, 60.0), 2)
device_type = random.choice(['mobile_phone', 'tablet', 'desktop'])
conversion_status = random.choice(['repeating user', 'new user'])
clicks_per_sesh = random.randint(1, 20)
listing_clicked = random.randint(0, 9)
booking_success = random.randint(0, 1)

if booking_success == 1:
# First, get the confirmed booking ID, regardless of payment status
cursor.execute("SELECT Bookings.booking_id FROM Bookings WHERE guest_id = ? AND
Bookings.booking_status = 'confirmed'", (user_id,))
booking_data = cursor.fetchall()

if booking_data:
booking_id = random.choice(booking_data)[0] # Get the booking_id
else:
booking_id = random.choice(confirmed_booking_ids) if confirmed_booking_ids else None

else:
booking_id = None # Reset booking_id if booking_success is 0

user_activity_data.append((session_id_counter, session_date, time_spent_page, device_type,
conversion_status, clicks_per_sesh, user_id, booking_success, listing_clicked,
booking_id))
session_id_counter += 1
```

```
try:
cursor.executemany("""
INSERT INTO User_activity (session_id, session_date, time_spent_page, device_type,
conversion_status, clicks_per_sesh, user_id, booking_success, listing_clicked, booking_id)
VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
""", user_activity_data)
conn.commit()
except sqlite3.IntegrityError:
print(f"Error inserting user: {user_activity_data}. user_activity_id might already exist.")
except Exception as e:
print(f"An unexpected error occurred: {e}")
conn.close()

print("User_activity table populated with fake data.")
```

Part H: Amenities table

```
import sqlite3
import random

# Connect to the database
conn = sqlite3.connect("mydm.db")
cursor = conn.cursor()

# Create Amenities table
cursor.execute('''
CREATE TABLE IF NOT EXISTS Amenities (
amenities_id INTEGER PRIMARY KEY,
wifi TEXT CHECK (wifi IN ('yes_wifi', 'no_wifi')),
pet_friendly TEXT CHECK (pet_friendly IN ('yes_pet_friendly', 'no_pet_friendly')),
parking TEXT CHECK (parking IN ('yes_parking', 'no_parking')),
fireplace TEXT CHECK (fireplace IN ('yes_fireplace', 'no_fireplace')),
swimming_pool TEXT CHECK (swimming_pool IN ('yes_swimming_pool', 'no_swimming_pool')),
tv TEXT CHECK (tv IN ('yes_tv', 'no_tv')),
kettle TEXT CHECK (kettle IN ('yes_kettle', 'no_kettle')),
fridge TEXT CHECK (fridge IN ('yes_fridge', 'no_fridge')),
induction TEXT CHECK (induction IN ('yes_induction', 'no_induction')),
microwave TEXT CHECK (microwave IN ('yes_microwave', 'no_microwave')),
towels TEXT CHECK (towels IN ('yes_towels', 'no_towels')),
laundry TEXT CHECK (laundry IN ('yes_laundry', 'no_laundry')),
secu_camera TEXT CHECK (secu_camera IN ('yes_secu_camera', 'no_secu_camera')),
balcony TEXT CHECK (balcony IN ('yes_balcony', 'no_balcony'))
);
''')
#0:not available 1:available

# Generate amenities data
amenities_data = []
for amenities_id in range(1,501): # Adjust the range to control the number of amenities entries
amenities_data.append((
amenities_id,
random.choice(['yes_wifi', 'no_wifi']),
random.choice(['yes_pet_friendly', 'no_pet_friendly']),
random.choice(['yes_parking', 'no_parking']),
random.choice(['yes_fireplace', 'no_fireplace']),
random.choice(['yes_swimming_pool', 'no_swimming_pool']),
random.choice(['yes_tv', 'no_tv']),
random.choice(['yes_kettle', 'no_kettle']),
random.choice(['yes_fridge', 'no_fridge']),
random.choice(['yes_induction', 'no_induction']),
random.choice(['yes_microwave', 'no_microwave']),
random.choice(['yes_towels', 'no_towels']),
random.choice(['yes_laundry', 'no_laundry']),
random.choice(['yes_secu_camera', 'no_secu_camera']),
random.choice(['yes_balcony', 'no_balcony'])
))

# Insert amenities data
try:
cursor.executemany("""
INSERT INTO Amenities (amenities_id, wifi, pet_friendly, parking, fireplace, swimming_pool, tv,
kettle, fridge, induction, microwave, towels, laundry, secu_camera, balcony)
VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
""", amenities_data)
conn.commit()
```

```
except sqlite3.IntegrityError:
print(f"Error inserting user: {amenities_data}. amenities_id might already exist.")
except Exception as e:
print(f"An unexpected error occurred: {e}")

# Close the connection
conn.close()

print("Amenities table populated with fake data.")
```

Part I: Refunds table

```
import sqlite3
import random
from faker import Faker
from datetime import datetime, timedelta

# Initialise Faker
fake = Faker()

# Connect to the database
conn = sqlite3.connect("mydm.db")
cursor = conn.cursor()

# Create Refunds table
cursor.execute('''
CREATE TABLE IF NOT EXISTS Refunds (
refund_id INTEGER PRIMARY KEY,
refund_type TEXT CHECK (refund_type IN ('full', 'partial', 'none')),
refund_status TEXT CHECK (refund_status IN ('pending', 'approved', 'rejected')),
refund_amount DECIMAL,
request_date TEXT,
processed_date TEXT,
booking_id INTEGER,
payment_id INTEGER,
FOREIGN KEY (booking_id) REFERENCES Bookings (booking_id),
FOREIGN KEY (payment_id) REFERENCES Payments (payment_id)
);
''')

# Get available booking IDs
cursor.execute("SELECT Bookings.booking_id, Bookings.booking_date, Payments.payment_id,
Payments.payment_amount FROM Bookings INNER JOIN Payments ON Bookings.booking_id =
Payments.booking_id")
available_booking_payment_ids = cursor.fetchall()

# Generate refunds data
refunds_data = []
refund_id_counter = 1

for booking_id, booking_date, payment_id, payment_amount in available_booking_payment_ids:
if random.random() < 0.3:
refund_type = random.choice(['full', 'partial', 'none'])
# refund_status = 'pending' # Default to pending
refund_amount = 0 # Default to 0
processed_date_str = None # Default to None

booking_date = datetime.strptime(booking_date_str, "%Y-%m-%d").date()
request_date = fake.date_between(start_date=booking_date + timedelta(days=1), end_date =
booking_date + timedelta(days=14))
request_date_str = request_date.strftime("%Y-%m-%d") if isinstance(request_date, datetime)
else request_date

if refund_type == 'none':
refund_status = 'rejected' # Directly set to rejected
processed_date = fake.date_between(start_date=request_date, end_date=request_date +
timedelta(days=14))
processed_date_str = processed_date.strftime("%Y-%m-%d") if isinstance(processed_date,
datetime) else processed_date
elif refund_type == 'full' and booking_status =='cancelled by host':
refund_status = 'approved'
refund_amount = payment_amount
processed_date = fake.date_between(start_date=request_date, end_date=request_date +
timedelta(days=14))
```

```
processed_date_str = processed_date.strftime("%Y-%m-%d") if isinstance(processed_date,
datetime) else processed_date
elif refund_type == 'partial': # and booking_status in ('cancelled by guest')
refund_status = 'approved'
refund_amount = payment_amount * random.uniform(0.1, 0.9) # Some percentage of payment
processed_date = fake.date_between(start_date=request_date, end_date=request_date +
timedelta(days=14))
processed_date_str = processed_date.strftime("%Y-%m-%d") if isinstance(processed_date,
datetime) else processed_date
else:
# For all other cases, it's either rejected or pending with a partial refund.
refund_status = random.choice(['pending', 'rejected'])
if refund_status == 'rejected':
processed_date = fake.date_between(start_date=request_date, end_date=request_date +
timedelta(days=14))
processed_date_str = processed_date.strftime("%Y-%m-%d") if isinstance(processed_date,
datetime) else processed_date

refunds_data.append((refund_id_counter, refund_type, refund_status, refund_amount,
request_date_str, processed_date_str, booking_id, payment_id))
refund_id_counter += 1



# Insert refunds data
try:
cursor.executemany("""
INSERT INTO Refunds (refund_id, refund_type, refund_status, refund_amount, request_date,
processed_date, booking_id, payment_id)
VALUES (?, ?, ?, ?, ?, ?, ?, ?)
""", refunds_data)
conn.commit()
except sqlite3.IntegrityError:
print(f"Error inserting user: {refunds_data}. refund_id might already exist.")
except Exception as e:
print(f"An unexpected error occurred: {e}")

# Close the connection
```

```
conn.close()

print("Refunds table populated with fake data.")
```

Part J: Listing_amenities table

```
#THIS WOULD BE TO GET 1 SET OF AMENITIES PER LISTING
import sqlite3
import random

# Connect to the database
conn = sqlite3.connect("mydm.db")
cursor = conn.cursor()

# Create Listing_amenities table
cursor.execute('''
CREATE TABLE IF NOT EXISTS Listing_amenities (
listings_id INTEGER NOT NULL PRIMARY KEY,
amenities_id INTEGER NOT NULL,
FOREIGN KEY (listings_id) REFERENCES Listings (listing_id),
FOREIGN KEY (amenities_id) REFERENCES Amenities (amenities_id)
);
''')

# Get available listings_id and amenities_id values
cursor.execute("SELECT listing_id FROM Listings")
available_listings_ids = [row[0] for row in cursor.fetchall()]
cursor.execute("SELECT amenities_id FROM Amenities")
available_amenities_ids = [row[0] for row in cursor.fetchall()]

# Generate Listing_amenities data (modified)
listing_amenities_data = []
for listing_id in available_listings_ids:
# Randomly select one amenities_id for this listing
amenities_id = random.choice(available_amenities_ids)
listing_amenities_data.append((listing_id, amenities_id))

# Insert Listing_amenities data
```

```
try:
cursor.executemany("""
INSERT INTO Listing_amenities (listings_id, amenities_id)
VALUES (?, ?)
""", listing_amenities_data)
conn.commit()
except sqlite3.IntegrityError:
print(f"Error inserting: {listing_amenities_data}. listings_id might already exist.")
except Exception as e:
print(f"An unexpected error occurred: {e}")

# Close the connection
conn.close()

print("Listing_amenities table populated with fake data.")
```

Appendix B: Validation and accuracy checks of the generated data for UKStay

```
import sqlite3

# Connect to the database
conn = sqlite3.connect("mydm.db")
cursor = conn.cursor()

#data integrity checks
def check_foreign_keys():
#Check if foreign key references are valid
cursor.execute("SELECT COUNT(*) FROM Users WHERE cities_id NOT IN (SELECT cities_id
FROM UK_cities)")
invalid_users = cursor.fetchone()[0]

cursor.execute("SELECT COUNT(*) FROM Listings WHERE host_id NOT IN (SELECT user_id
FROM Users)")
invalid_listings = cursor.fetchone()[0]

print(f"Foreign Key Issues: {invalid_users} invalid Users, {invalid_listings} invalid Listings")
```

```
def check_unique_constraints():
#Ensure unique constraints hold, such as unique emails
cursor.execute("SELECT email, COUNT(*) FROM Users GROUP BY email HAVING
COUNT(*) > 1")
duplicates = cursor.fetchall()
print(f"Duplicate Emails: {len(duplicates)}")

#logical data validation
def check_logical_constraints():
#Check logical constraints like price, minimum nights, and valid dates
cursor.execute("SELECT COUNT(*) FROM Listings WHERE price_per_night <= 0")
invalid_prices = cursor.fetchone()[0]

cursor.execute("SELECT COUNT(*) FROM Listings WHERE min_nights < 1")
invalid_nights = cursor.fetchone()[0]

cursor.execute("SELECT COUNT(*) FROM Listings WHERE listing_date > DATE('now')")
future_dates = cursor.fetchone()[0]

print(f"Logical Errors: {invalid_prices} invalid Prices, {invalid_nights} invalid Min Nights,
{future_dates} future Dates")

def check_booking_constraints():
# Check if check-out date is after check-in date
cursor.execute("SELECT COUNT(*) FROM Bookings WHERE check_out_date <=
check_in_date")
invalid_booking_dates = cursor.fetchone()[0]

# Check if booking date is before check-in date
cursor.execute("SELECT COUNT(*) FROM Bookings WHERE booking_date > check_in_date")
invalid_booking_date_order = cursor.fetchone()[0]

print(f"Booking Constraints: {invalid_booking_dates} invalid booking date ranges,
{invalid_booking_date_order} invalid booking date order")

def check_payment_constraints():
# Check if payment amount is positive
```

```
cursor.execute("SELECT COUNT(*) FROM Payments WHERE payment_amount <= 0")
invalid_payment_amounts = cursor.fetchone()[0]

print(f"Payment Constraints: {invalid_payment_amounts} invalid payment amounts")

def check_review_constraints():
# Check if rating score is within the valid range (1-5)
cursor.execute("SELECT COUNT(*) FROM Reviews WHERE rating_score < 1 OR
rating_score > 5")
invalid_rating_scores = cursor.fetchone()[0]

print(f"Reviews Constraints: {invalid_rating_scores} invalid rating scores")

#cross-table consistency checks
def check_inconsistent_payment_consistency():
# Check if total payment amount matches the sum of individual booking payments
cursor.execute("""
SELECT COUNT(*)
FROM Bookings B
JOIN Payments P ON B.booking_id = P.booking_id
JOIN Listings L ON B.listing_id = L.listing_id -- Join with Listings to get price_per_night
WHERE P.payment_amount != (L.price_per_night * B.num_guests)
""")
inconsistent_payments = cursor.fetchone()[0]
print(f"Cross-Table Issues: {inconsistent_payments} inconsistent payments")

def check_listing_availability():
#check that bookings are not made on listings that aren't available
cursor.execute("""
SELECT COUNT(*)
FROM Bookings B
JOIN Listings L ON B.listing_id = L.listing_id
WHERE L.availability_status = 'booked'
AND B.check_in_date < L.listing_date -- Assuming listing_date is the date when it becomes
available
AND B.check_out_date > L.listing_date
""")
```

```
inconsistent_bookings = cursor.fetchone()[0]
print(f"Cross-Table Issues: {inconsistent_bookings} bookings for unavailable listings.")

def check_refund_payment_consistency():
#ensure that the refund amount does not exceed the payment amount
cursor.execute("""
SELECT COUNT(*)
FROM Refunds R
JOIN Payments P ON R.payment_id = P.payment_id
WHERE R.refund_amount > P.payment_amount -- Check for over-refunding
""")
over_refunds = cursor.fetchone()[0]
print(f"Cross-Table Issues: {over_refunds} refunds exceeding payment amounts.")

#business rule compliance
def check_booking_rules():
# Check if bookings for the same listing overlap (double-booking)
cursor.execute("""
SELECT COUNT(*)
FROM Bookings B1, Bookings B2
WHERE B1.listing_id = B2.listing_id
AND B1.booking_id != B2.booking_id
AND B1.check_in_date < B2.check_out_date
AND B1.check_out_date > B2.check_in_date
""")
double_bookings = cursor.fetchone()[0]
print(f"Bookings Rule Violations: {double_bookings} double bookings detected")

def check_listing_user_rules():
# Listings posted by 'Guest' user type
cursor.execute("""
SELECT COUNT(*)
FROM Listings L
JOIN Users U ON L.host_id = U.user_id
WHERE U.user_type = 'Guest'
""")
guest_listings = cursor.fetchone()[0]
```

```
print(f"Listing User Rule Violations: {guest_listings} listings posted by 'Guest' users.")

def check_refund_payment_rules():
#check that each refund is associated with a payment entry
cursor.execute("""
SELECT COUNT(*)
FROM Payments P
WHERE NOT EXISTS (SELECT 1 FROM Refunds R WHERE P.payment_id = R.payment_id)
AND (booking_id, payment_amount) in (SELECT booking_id, payment_amount FROM Bookings
WHERE payment_amount >0 AND check_in_date < date() AND booking_status in ('cancelled by
host', 'cancelled by guest'))
""")
refund_missing_payments = cursor.fetchone()[0]
print(f"Refund Payment Rule Violations: {refund_missing_payments} refund payment entries are
missing.")

def run_validity_checks():
print("Running validity checks...")
check_foreign_keys()
check_unique_constraints()
check_logical_constraints()
check_booking_constraints()
check_payment_constraints()
check_review_constraints()
check_inconsistent_payment_consistency()
check_listing_availability()
check_refund_payment_consistency()
check_booking_rules()
check_listing_user_rules()
check_refund_payment_rules()
print("Validity checks completed.")

run_validity_checks()

# Close the connection
conn.close()
```

Appendix C: SQL queries generating UKStay's business insights reports

Part A: Import Fake data and database

```
import pandas as pd
import sqlite3

conn = sqlite3.connect('mydm-3.db')

# 1.
bookings_df = pd.read_csv("Bookings.csv", index_col=0)
print("\nBookings Dataset   : ", bookings_df.columns.tolist())

# 2.
payments_df = pd.read_csv("Payments.csv", index_col=0)
print("\nPayments Dataset   : ", payments_df.columns.tolist())

# 3.
refunds_df = pd.read_csv("Refunds.csv", index_col=0)
print("\nRefunds Dataset   : ", refunds_df.columns.tolist())

# 4.
reviews_df = pd.read_csv("Reviews.csv", index_col=0)
print("\nReviews Dataset   : ", reviews_df.columns.tolist())

# 5.
user_activity_df = pd.read_csv("User_activity.csv", index_col=0)
print("\nUser Activity Dataset   : ", user_activity_df.columns.tolist())

# 6.
users_df = pd.read_csv("Users.csv", index_col=0)
print("\nUsers Dataset   : ", users_df.columns.tolist())

# 7.
cities_df = pd.read_csv("UK_cities.csv", index_col=0)
print("\nUK Cities Dataset   : ", cities_df.columns.tolist())

# 8. house data
listings_df = pd.read_csv("Listings.csv", index_col=0)
```

```
print("\nListings Dataset  : ", listings_df.columns.tolist())

# 9. lising amenities data
listing_amenities_df = pd.read_csv("Listing_amenities.csv", index_col=0)
print("\nListing Amenities Dataset  : ", listing_amenities_df.columns.tolist())

# 10.
amenities_df = pd.read_csv("Amenities.csv", index_col=0)
print("\nAmenities Dataset  : ", amenities_df.columns.tolist())
```

Part B.a: Some steps to verify the data validity

```
import sqlite3

conn = sqlite3.connect('mydm-3.db')

# Insert DataFrame data into SQLite database
cities_df.to_sql('UK_cities', conn, if_exists='append', index=False)
users_df.to_sql('Users', conn, if_exists='append', index=False)
listings_df.to_sql('Listings', conn, if_exists='append', index=False)
bookings_df.to_sql('Bookings', conn, if_exists='append', index=False)
payments_df.to_sql('Payments', conn, if_exists='append', index=False)
reviews_df.to_sql('Reviews', conn, if_exists='append', index=False)
refunds_df.to_sql('Refunds', conn, if_exists='append', index=False)
user_activity_df.to_sql('User_activity', conn, if_exists='append', index=False)
amenities_df.to_sql('Amenities', conn, if_exists='append', index=False)
listing_amenities_df.to_sql('Listing_amenities', conn, Roboto if_exists='append', index=False)
```

Part B.b.: Ensure database integrity

```
conn = sqlite3.connect("mydm-3.db")
cursor = conn.cursor()

cursor.execute("VACUUM;")
print("database has been fixed 。")
```

Part B.c.: Check the integrity of the key

```
import sqlite3
import pandas as pd



conn = sqlite3.connect('mydm-3.db')
cursor = conn.cursor()



cursor.execute("SELECT check_in_date FROM bookings LIMIT 5;")
sample_dates = cursor.fetchall()
print("date format check : ", sample_dates)



def check_foreign_key(child_table, child_col, parent_table, parent_col):
cursor.execute(f"""
SELECT COUNT(*) FROM {child_table}
WHERE {child_col} NOT IN (SELECT {parent_col} FROM {parent_table})
""")
invalid = cursor.fetchone()[0]
print(f"{child_table}.{child_col} invalid number is : {invalid}")

check_foreign_key("listings", "cities_id", "UK_cities", "cities_id")
check_foreign_key("bookings", "listing_id", "listings", "listing_id")
check_foreign_key("payments", "booking_id", "bookings", "booking_id")
```

Part C: Start to analyse

Part C.a.: Analyze the overall performance of each city and the sales data of confirmed orders (ranked by Revenue)

```
import pandas as pd

def analyze_sales_by_city():
query = """
WITH paid_payments AS (
-- only for number of paid
SELECT
booking_id,
SUM(payment_amount) AS total_payment
FROM payments
WHERE payment_status = 'paid' -- Only count successful payment records
GROUP BY booking_id
),
---Calculate the total refund amount for each order
refund_agg AS (
SELECT
booking_id,
SUM(refund_amount) AS total_refund
FROM refunds
GROUP BY booking_id
)
SELECT
c.cities_name AS city,
COUNT(DISTINCT b.booking_id) AS confirmed_bookings, -- Confirmed Orders
COALESCE(SUM(p.total_payment), 0) - COALESCE(SUM(r.total_refund), 0) AS net_revenue,
(COALESCE(SUM(p.total_payment), 0) - COALESCE(SUM(r.total_refund), 0))
/ NULLIF(COUNT(DISTINCT b.booking_id), 0) AS avg_booking_value
FROM bookings b
JOIN listings l ON b.listing_id = l.listing_id
JOIN UK_cities c ON l.cities_id = c.cities_id
LEFT JOIN paid_payments p ON b.booking_id = p.booking_id -- connect to paid
LEFT JOIN refund_agg r ON b.booking_id = r.booking_id -- connect to refund
WHERE p.booking_id IS NOT NULL -- only payment has been confirmed
GROUP BY city
ORDER BY net_revenue DESC
"""
cursor.execute(query)
df = pd.DataFrame(
cursor.fetchall(),
columns=["city", "confirmed_bookings", "net_revenue", "avg_booking_value"]
)
return df

def analyze_city_performance():
"""
```

```
"""
query = """
WITH all_payments AS (
-- （paid + pending）
SELECT
booking_id,
SUM(payment_amount) AS total_payment
FROM payments
GROUP BY booking_id
)
SELECT
c.cities_name AS city,
COUNT(DISTINCT b.booking_id) AS total_bookings, -- total bookings
COALESCE(SUM(p.total_payment), 0) AS total_revenue -- total revenue）
FROM bookings b
JOIN listings l ON b.listing_id = l.listing_id
JOIN UK_cities c ON l.cities_id = c.cities_id
LEFT JOIN all_payments p ON b.booking_id = p.booking_id
GROUP BY c.cities_name
ORDER BY total_revenue DESC
"""
cursor.execute(query)
df = pd.DataFrame(
cursor.fetchall(),
columns=["city", "total_bookings", "total_revenue"]
)
return df


sales_by_city = analyze_sales_by_city()
city_perf = analyze_city_performance()

merged_df = pd.merge(
city_perf[["city", "total_bookings", "total_revenue"]],
sales_by_city[["city", "confirmed_bookings", "net_revenue", "avg_booking_value"]],
on="city",
how="outer" #
).fillna(0)

merged_df["loss"] = merged_df["total_revenue"] - merged_df["net_revenue"]

merged_df = merged_df[[
"city",
"total_bookings",
"confirmed_bookings",
"avg_booking_value",
"total_revenue",
"net_revenue",
"loss"
]]

merged_df = merged_df.sort_values(by="net_revenue", ascending=False)

print("=== Booking Distribution by City ===")
print(merged_df.head(10))
```

Part C.b.: Analyze the overall performance of each city and the sales data of confirmed orders (ranked by bookings)

```
import pandas as pd

def analyze_sales_by_city():
"""


"""
query = """
WITH paid_payments AS (
-- only for number of paid
SELECT
booking_id,
SUM(payment_amount) AS total_payment
FROM payments
WHERE payment_status = 'paid'
GROUP BY booking_id
```

```
),
refund_agg AS (
SELECT
booking_id,
SUM(refund_amount) AS total_refund
FROM refunds
GROUP BY booking_id
)
SELECT
c.cities_name AS city,
COUNT(DISTINCT b.booking_id) AS confirmed_bookings,
COALESCE(SUM(p.total_payment), 0) - COALESCE(SUM(r.total_refund), 0) AS net_revenue,
(COALESCE(SUM(p.total_payment), 0) - COALESCE(SUM(r.total_refund), 0))
/ NULLIF(COUNT(DISTINCT b.booking_id), 0) AS avg_booking_value
FROM bookings b
JOIN listings l ON b.listing_id = l.listing_id
JOIN UK_cities c ON l.cities_id = c.cities_id
LEFT JOIN paid_payments p ON b.booking_id = p.booking_id -- connect to paid
LEFT JOIN refund_agg r ON b.booking_id = r.booking_id -- connect to refund
WHERE p.booking_id IS NOT NULL -- only payment has been confirmed
GROUP BY city
ORDER BY net_revenue DESC
"""
cursor.execute(query)
df = pd.DataFrame(
cursor.fetchall(),
columns=["city", "confirmed_bookings", "net_revenue", "avg_booking_value"]
)
return df

def analyze_city_performance():
"""


"""
query = """
WITH all_payments AS (
-- （paid + pending）
```

```
SELECT
booking_id,
SUM(payment_amount) AS total_payment
FROM payments
GROUP BY booking_id
)
SELECT
c.cities_name AS city,
COUNT(DISTINCT b.booking_id) AS total_bookings, -- total bookings
COALESCE(SUM(p.total_payment), 0) AS total_revenue -- total revenue）
FROM bookings b
JOIN listings l ON b.listing_id = l.listing_id
JOIN UK_cities c ON l.cities_id = c.cities_id
LEFT JOIN all_payments p ON b.booking_id = p.booking_id
GROUP BY c.cities_name
ORDER BY total_revenue DESC
"""
cursor.execute(query)
df = pd.DataFrame(
cursor.fetchall(),
columns=["city", "total_bookings", "total_revenue"]
)
return df


sales_by_city = analyze_sales_by_city()
city_perf = analyze_city_performance()

merged_df = pd.merge(
city_perf[["city", "total_bookings", "total_revenue"]],
sales_by_city[["city", "confirmed_bookings", "net_revenue", "avg_booking_value"]],
on="city",
how="outer" #
).fillna(0)

merged_df["loss"] = merged_df["total_revenue"] - merged_df["net_revenue"]
```

```
merged_df = merged_df[[
"city",
"total_bookings",
"confirmed_bookings",
"avg_booking_value",
"total_revenue",
"net_revenue",
"loss"
]]

merged_df = merged_df.sort_values(by="confirmed_bookings", ascending=False)

print("=== Booking Distribution by City ===")
print(merged_df.head(10))
```

Part C.c.: Monthly booking trend analysis (region distribution)

```
def analyze_monthly_trends():
query = """
SELECT
strftime('%Y-%m', b.check_in_date) AS month,
c.cities_name AS city,
COUNT(b.booking_id) AS booking_count,
SUM(p.payment_amount) AS monthly_revenue
FROM bookings b
JOIN listings l ON b.listing_id = l.listing_id
JOIN UK_cities c ON l.cities_id = c.cities_id
JOIN payments p ON b.booking_id = p.booking_id
GROUP BY month, city
ORDER BY month, monthly_revenue DESC
"""
cursor.execute(query)
df = pd.DataFrame(cursor.fetchall(), columns=["month", "city", "bookings", "revenue"])
return df



monthly_trends = analyze_monthly_trends()
pivot_table = monthly_trends.pivot(index="month", columns="city", values="bookings").fillna(0)
```

```
print("\n=== Monthly regional booking trends===")
print(pivot_table.head())
```

Part C.d.: Monthly booking trend

```
def analyze_monthly_trends():
query = """
-- Add EXPLAIN QUERY PLAN to analyze execution path
EXPLAIN QUERY PLAN
SELECT
strftime('%Y-%m', b.check_in_date) AS month,
c.cities_name AS city,
COUNT(b.booking_id) AS booking_count,
SUM(p.payment_amount) AS total_revenue
FROM bookings b
JOIN listings l ON b.listing_id = l.listing_id
JOIN UK_cities c ON l.cities_id = c.cities_id
JOIN payments p ON b.booking_id = p.booking_id
GROUP BY month, city
ORDER BY month, city
"""



# Actually executing the query
cursor.execute(query.replace("EXPLAIN QUERY PLAN", ""))
df = pd.DataFrame(cursor.fetchall(), columns=["month", "city", "bookings", "revenue"])
return df



monthly_trends = analyze_monthly_trends()
print("\n=== monthly trend ===")
print(monthly_trends.head(50))
```

Part D: Popularity of property type analysis

Part D.a.: Ranked by revenue

```
def analyze_property_performance():
query = """
```

```
SELECT
l.property_type,
COUNT(DISTINCT CASE WHEN p.payment_status = 'paid' THEN b.booking_id END) AS
paid_bookings,
SUM(p.payment_amount) AS total_revenue,
SUM(CASE WHEN p.payment_status = 'paid' THEN p.payment_amount ELSE 0 END) AS
paid_revenue,
AVG(l.price_per_night) AS avg_price,
(COUNT(DISTINCT CASE WHEN p.payment_status = 'paid' THEN b.booking_id END) * 1.0
/ (SELECT COUNT(DISTINCT booking_id) FROM payments WHERE payment_status = 'paid'))
AS market_share
FROM bookings b
JOIN listings l ON b.listing_id = l.listing_id
JOIN payments p ON b.booking_id = p.booking_id
GROUP BY l.property_type
ORDER BY total_revenue DESC
"""
cursor.execute(query)
df = pd.DataFrame(
cursor.fetchall(),
columns=["property_type", "paid_bookings", "total_revenue", "NET_revenue", "avg_price",
"market_share"]
)
return df

property_perf = analyze_property_performance()
print("\n=== property type analysis===")
print(property_perf)
```

Part D.b.: Ranked by booking number

```
def analyze_property_performance():
query = """
SELECT
l.property_type,
COUNT(DISTINCT CASE WHEN p.payment_status = 'paid' THEN b.booking_id END) AS
bookings,
SUM(p.payment_amount) AS total_revenue,
SUM(CASE WHEN p.payment_status = 'paid' THEN p.payment_amount ELSE 0 END) AS
paid_revenue,
AVG(l.price_per_night) AS avg_price,
(COUNT(DISTINCT CASE WHEN p.payment_status = 'paid' THEN b.booking_id END) * 1.0
/ (SELECT COUNT(DISTINCT booking_id) FROM payments WHERE payment_status = 'paid'))
AS market_share
FROM bookings b
JOIN listings l ON b.listing_id = l.listing_id
JOIN payments p ON b.booking_id = p.booking_id
GROUP BY l.property_type
ORDER BY bookings DESC
"""
cursor.execute(query)
df = pd.DataFrame(
cursor.fetchall(),
columns=["property_type", "bookings", "total_revenue", "NET_revenue", "avg_price",
"market_share"]
)
return df

property_perf = analyze_property_performance()
print("\n=== property type analysis） ===")
print(property_perf)
```

Part E: Demand-Supply Ratios calculation

```
def analyze_demand_supply_ratio():
query = """
SELECT
c.cities_name AS city,
l.property_type,
COUNT(CASE WHEN b.booking_status = 'confirmed' THEN b.booking_id END) AS demand,
COUNT(l.listing_id) AS supply,
(COUNT(CASE WHEN b.booking_status = 'confirmed' THEN b.booking_id END) * 1.0 /
NULLIF(COUNT(l.listing_id), 0)) AS demand_supply_ratio
FROM listings l
LEFT JOIN bookings b ON l.listing_id = b.listing_id
```

```
JOIN UK_cities c ON l.cities_id = c.cities_id
GROUP BY city, property_type
ORDER BY demand_supply_ratio DESC;
"""
cursor.execute(query)
df = pd.DataFrame(cursor.fetchall(), columns=["city", "property_type", "demand", "supply",
"demand_supply_ratio"])
return df


demand_supply_data = analyze_demand_supply_ratio()
print("\n=== Demand-Supply Ratio Analysis ===")
print(demand_supply_data)
print("\n=== TOP 10 Demand-Supply Ratio Analysis ===")
print(demand_supply_data.head(10))
```

Part F: Rating score of property type calculation

```
def calculate_property_type_ratings():
query = """
SELECT
l.property_type,
ROUND(AVG(r.rating_score), 2) AS avg_rating,
COUNT(r.rating_score) AS total_reviews
FROM bookings b
JOIN listings l ON b.listing_id = l.listing_id
JOIN payments p ON b.booking_id = p.booking_id
LEFT JOIN reviews r ON b.booking_id = r.booking_id
WHERE p.payment_status = 'paid'
GROUP BY l.property_type
ORDER BY avg_rating DESC
"""
cursor.execute(query)
results = cursor.fetchall()
df = pd.DataFrame(results, columns=["property_type", "avg_rating", "total_reviews"])
return df
```

```
try:
ratings_df = calculate_property_type_ratings()
print("=== Property type avg score ===")
print(ratings_df.head())
except Exception as e:
print("errors:", e)
```